

Coupling Static and Dynamic Semantics in an Architecture Description Language

Roshanak Roshandel Nenad Medvidovic

*Computer Science Department
University of Southern California
Los Angeles, CA 90089-0781, USA
{roshande, neno}@usc.edu*

Abstract

One of the problems in designing today's complex, reactive and mission critical systems is providing extensive multi-dimensional analysis of the systems at the architectural level. Architecture description languages (ADLs) provide such analysis capabilities. However, ADLs mainly focus on specific system aspects (e.g., formal refinement rules, deadlock detection, or schedulability analysis). In this paper, we describe an approach to combining the syntax and semantics of two heterogeneous architecture modeling notations in such a way that they remain consistent. The resulting integrated notation and toolset expand the available architecture modeling and analysis capabilities. In turn, this enables the architect to uncover a wider range of problems early in the development process and thus assists in reducing the development costs.

1. Introduction

One of the goals of software architecture research has been to equip developers with powerful representation and analysis tools, which allow the developers to effectively leverage high-level, formal architectural models in the engineering of large, complex software systems [22,25]. A number of architecture description languages (ADLs) have been proposed to provide just such modeling and analysis support [17]. These languages are usually characterized by formal syntax and semantics intended to facilitate the representation and analysis of *specific* aspects of a software system's architecture (e.g., Wright's support for deadlock detection [1] and UniCon's support for schedulability analysis [24]), possibly within the confines of a *specific* architectural style (e.g., C2SADEL's support for C2-style architectures [26] and GenVoca's support for layered architectures [2]).

While each such ADL is powerful in its own right and leverages its narrow focus to provide some sophisticated analysis capabilities, that narrow focus ultimately

hampers the ADL's general applicability. For example, no support currently exists for the schedulability analysis of C2-style architectures or deadlock analysis of GenVoca-style architectures. For this reason, several approaches have recently attempted to provide mechanisms for bridging different ADLs and enabling their use in tandem in a software development project: ACME [6], xADL [3,10], and UML [14]. However, for the most part these approaches have focused on bridging the divergent syntactic bases of the ADLs, while the issue of ADLs' semantics has not been explicitly and completely addressed to date. For example, ACME leverages *uninterpreted* property specifications to capture different ADLs' semantics; similarly, xADL provides a common basis for architectural description, but leaves the issue of architectural semantics to the engineers building the ADL-specific extensions upon that provided basis.

In this paper, we describe an approach to relating *both* the syntax and the semantics of two architecture modeling notations. We have developed mechanisms for coupling the modeling and analysis power of an existing ADL—C2SADEL [16]—and a well-known, general-purpose software modeling notation—StateCharts [7]. The underlying reason for selecting these two notations and the particular challenge we faced in relating them is that C2SADEL supports modeling of *static* semantics of a software system's architecture (via declarative behavioral “snapshots” expressed in component invariants and operation pre- and post-conditions), while StateCharts supports modeling of a system's *dynamic* semantics (via a continuum of system states and state transitions, events triggering those transitions, and actions caused by those events). Coupling C2SADEL and StateCharts allows us to augment the benefits of both respective notations, while eliminating their shortcomings. Using the two notations in tandem also allows architects to select the notation with which they have had more extensive experience. Furthermore, the combination of C2SADEL and StateCharts expands the range of possible architectural analyses. For example these analyses could be expanded by defining interaction protocols among components and studying their interactions for properties such as consistency and interaction path integrity.

Ultimately, this work will significantly enhance our already existing support for automatically generating partial implementations from architectural models [16]. Note that, unlike ACME, xADL, and UML, our approach is not intended as a general solution. Nonetheless, our approach provides a useful extension for modeling and analyzing architectures in a specific style (C2), and may serve to inform the on-going efforts centered on the above three approaches.

The remainder of this paper is organized as follows: In sections 2 and 3, we briefly present the background information and an example application that together set the stage for the ensuing discussion. Section 4, presents an overview of our static modeling notation, C2SADEL. Section 5, discusses our approach to coupling C2SADEL, and the dynamic modeling notation StateCharts. Section 6 discusses the implementation of our integrated tool support for static and dynamic architecture modeling and analysis. Finally section 7 presents our conclusions and proposes future work.

2. Background

To date, many architecture description languages (ADLs) have been developed to aid architecture-based system development [17]. ADLs provide formal notations for specifying software systems. Various tools for parsing, analysis, simulation, and code generation of the systems being modeled typically accompany these ADLs. Example ADLs include C2SADEL [16], Rapide [11], UniCon [24], Darwin [12], Wright [1], SADL [19], and ACME [6]. Some of these ADLs take system specification one step further by providing support for modeling behaviors of and constraints on architectural elements. These behaviors and constraints can be used to define conformance rules between services of interacting components (functional elements) and connectors (interaction elements) and thus ensure the consistency of an architecture throughout a system's lifespan. An ADL may be specifically tied to an architectural style [22,25], which is often suited to a particular problem domain.

Existing ADLs are typically narrowly focused on the specification and analysis of such architectural aspects as deadlock analysis, simulation, refinement, static structure, behavior, and so forth [17]. The ADLs leverage those narrow foci to provide some sophisticated analysis capabilities. At the same time, the narrow focus ultimately hampers a given ADL's general applicability. For this reason, several approaches have recently attempted to provide mechanisms for bridging different ADLs and enabling their combined use in a software development project: ACME [6], xADL [3,10], and UML [14]. However, for the most part these approaches have focused on bridging the divergent syntactic bases of the ADLs, while the issue of ADLs' semantics has not been

explicitly and completely addressed to date. For example, ACME leverages property specifications to capture different ADLs' semantics, but does not interpret those properties. Similarly, xADL leverages XML to provide a common basis for architectural description, but leaves the issue of architectural semantics to the engineers building each ADL-specific extension upon that basis. Finally, UML provides a set of modeling notations that have been shown to be effective in *capturing* the modeling characteristics of several ADLs [14]. However, the current work on representing ADL semantics in UML has not addressed the problem of *integrating* the different ADLs' divergent semantic bases.

In general, the different ADLs' approaches to architectural specification could be categorized as *static* and *dynamic*. Static specification techniques enable representation of system structure and possibly other aspects, such as behavior, performance, and distribution profile. The models of these system aspects are static if they declaratively specify properties that must hold at *discrete* points during the system's life span, but do not provide details on *how* those properties are to be achieved. Examples are UniCon's representation of the scheduling aspects of a component and C2SADEL's use of component invariants. On the other hand, dynamic modeling enables an architect to provide a more *continuous* view of how to arrive at a given property or desired state during a system's life span.¹ Examples of ADLs that provide such dynamic facilities for modeling architectural semantics include Wright [1] and Rapide [11]. Wright leverages CSP for representing component interactions and detecting deadlocks in a system, while Rapide employs partially ordered events to model and simulate the behaviors of distributed architectures. StateCharts [7] is another example notation widely used for describing dynamic system behaviors.

One useful coupling of ADLs would be to relate them such that the combined model notation and semantic basis provide the architect with extensive means for static as well as dynamic modeling and analysis. Such capabilities are currently not widely provided by ADLs. This deficiency is particularly notable in designing today's complex, reactive, and mission critical systems. Although static description of architectures and components is crucial for system design, dynamic behavioral specification provides more extensive analytic capabilities. In this paper, we attempt to remedy this shortcoming: we present a mechanism for coupling a static, component- and connector-based architectural description with StateCharts. Coupling the two notations enables us to leverage the particular capabilities of the

¹. In this paper, "dynamism" refers to the run-time behavioral semantics of a given architecture, rather than the dynamic reconfiguring of the architectural topology.

two methods and model a system more completely. It also helps provide additional analyses, which will help in detecting a wider range of defects early in the development process.

data between them, analyze the deployment, and, if desired, simulate the battle.

The application provides a view of the battlefield for one general and several commanders. The narrow dark

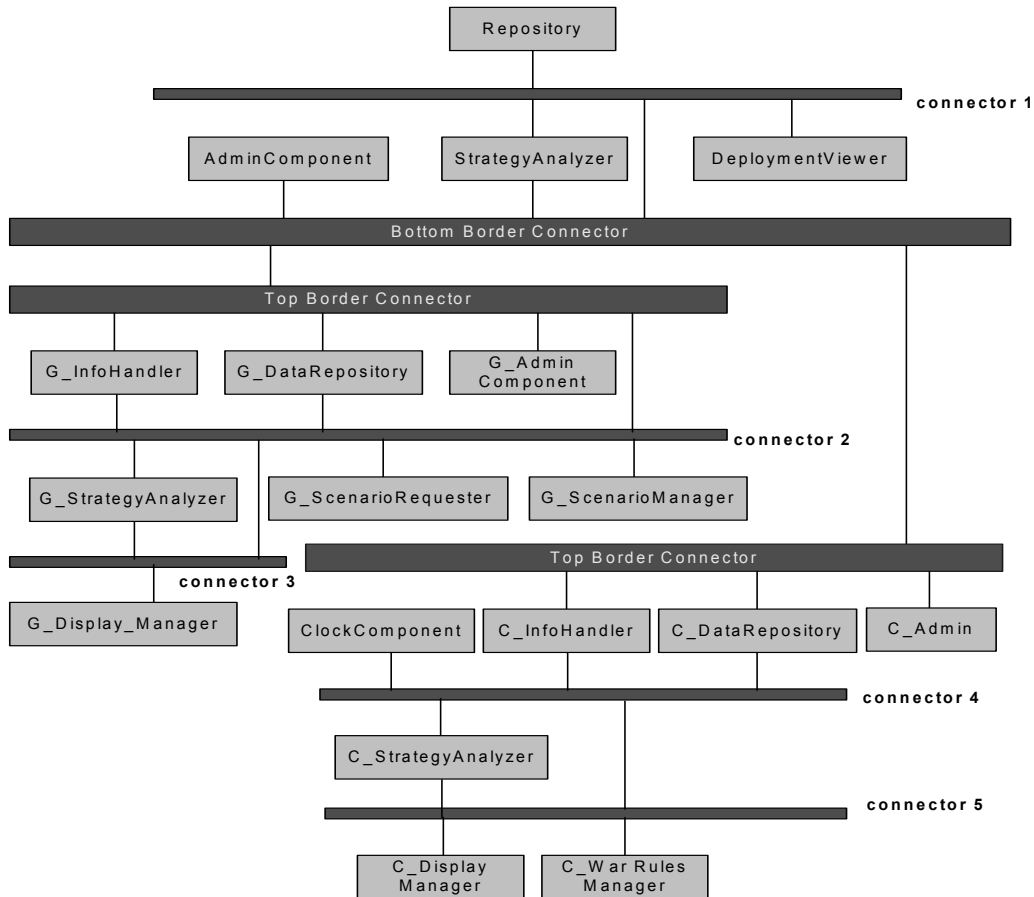


Figure 1. C2-style architecture of the Troops Deployment System

3. Example Application

Throughout this paper we will use a Troops Deployment System (TDS) as an example application. Figure 1 shows the architecture of the example system given in the C2 architectural style [26]. This architecture is an instance of an application family architecture targeted for distributed, run-time deployment of personnel intended to deal with situations such as natural disasters, military crises, and search-and-rescue efforts. The specific instance of this architecture family we are using in this paper addresses the deployment of military troops. Generals and commanders are deciding where to deploy troops based on the configuration of the terrain, the position of enemy troops and their goals. They can input the desired deployment configuration, communicate the

boxes in Figure 1 depict connectors while lighted color boxed show components in the system. The *border connectors* as labeled show the physical boundaries of multiple hardware platforms on which the application is running. Parts of the application below the two *Top Border Connectors* are replicated based on the number of commanders and generals in the system. The generals can view and deploy troops in the entire battlefield, while commanders are responsible for a particular segment of the field. Each general or commander can change their troops' arrangement in the field. The *DisplayManager* components send the new configurations to the local *DataRepository* components on a given device. The new configurations are propagated to the central ("headquarters") *Repository* component, which assembles the view of the complete battlefield and informs the interested parties based on the complete view of the field. A simulation of the battlefield based on the distribution

on the terrain, enemy troops, and mines is done using the *StrategyAnalyzer* components and its rules.

In this paper, we use a subset of TDS components and connectors for illustration purposes. Among the components and connectors in the system, we will specifically focus on *StrategyAnalyzer* and *Repository* components and *Connector 1*, which facilitates the communication between the two components. The *StrategyAnalyzer* component checks the deployment of the troops with respect to the location of the enemy troops and other obstacles (e.g., mines) and, based on the information received from the *Repository*, decides whether the configuration is acceptable or not.

4. C2SADEL: An Approach to Static Modeling

An architecture is specified in C2 style in three parts: component types, connector types, and topology. The topology, in turn, defines component and connector instances for a given system and their interconnections. A partial C2SADEL description of the TDS architecture shown in Figure 1 is given in Figure 2. The *StrategyAnalyzer* component type is specified *externally*, i.e., in a different file (*StrategyAnalyzer.c2*). The *AdminComponent* component type is specified as a *virtual* type: it can be used in the definition of the topology, but it does not have a specification and does not affect the behavioral analysis of the architecture. The concept of virtual types is useful in the case of components, such as *AdminComponent*, for which implementations are known to already exist, but which are not specified in C2SADEL.

Each C2SADEL component has a *name*, a set of *interface elements*, an associated *behavior*, and (possibly) an *implementation*. Each interface element has a direction indicator (*provided* or *required*), a name, a set of parameters, and (possibly) a result. Each parameter, in turn, has a name and a type. A component's behavior consists of an *invariant* and a set of *operations*. The invariant is used to specify properties that must be true of all component states. Each operation has pre-conditions, post-conditions, and (possibly) a result. Like interface elements, operations can be *provided* or *required*. Only provided operations will have an implementation in a given component. The pre- and post-conditions of required operations express their *expected* semantics.

Since we separate the interface from the behavior, we define a mapping function from interface elements to operations. This function is a total surjection: each interface element is mapped to a single operation, while each operation implements at least one interface. An interface element can be mapped to an operation only if the types of its parameters are subtypes of the

```
architecture TroopsDeploymentSystem is {
  component_types {
    component StrategyAnalyzer is extern
      {StrategyAnalyzer.C2;}
    component AdminComponent is virtual {}
  }
  connector_types {
    connector FiltConn is {filter msg_filter; }
    connector RegularConn is {filter no_filter;}
  }
  architectural_topology {
    component_instances {
      st_Analyzer: StrategyAnalyzer;
      admin: AdminComponent;
    }
    connector_instances {
      Connector1: RegularConn;
      BBC: FilteringConn;
    }
    connections {
      connector BBC {
        top
          st_Analyzer, admin;
        bottom
          TBC;
      }
      connector Connector1 {
        top
          Repository;
        bottom
          st_Analyzer, admin, dep_viewer;
      }
    }
  }
}
```

Figure 2. Partial TDS architecture specified in C2SADEL

corresponding variable types in the operation, while the type of its result is a supertype of operation's result type [15,16]. This property directly enables a single operation to export multiple interfaces. An interface element and its corresponding operation comprise a *component service*.

A partial specification of the *StrategyAnalyzer* component is given in Figure 4. Component invariant and operation pre- and post-conditions in are specified as first-order logic expressions. The *StrategyAnalyzer*'s invariant specifies that acceptable values for the state variable *status* are between 0 and 4, non-inclusive, throughout the system (1, 2, and 3 stand for initial, requested and done status values, respectively), and the values for *row* and *col* variables are bounded by a pixel location. Examples of a provided and a required operation are given. The required *orMapResult* operation should be provided by the *Repository* component in the *TDS* architecture, such that *StrategyAnalyzer*'s variable placeholder *PH* can be unified with the appropriate *theMap* state variable to satisfy the post-condition of the operation. As discussed above, *StrategyAnalyzer*'s interface is separated from its behavior and a surjective function is provided to map between the two.

5. Augmenting Static and Dynamic Modeling

In Section 2, we argued for coupling static and dynamic modeling techniques to expand the scope of architectural analysis. Static architectural specification such as that produced by C2SADEL is essential to design the topology of the system, separate concerns by creating components and connectors, and specify their behavioral “snapshots” at well-defined points in the modeled system’s execution. However, static semantics are incapable of sufficiently describing component behavior at run-time.

In order to mitigate the shortcoming of this type of architectural specification, we have developed a technique to couple the static specifications supported by C2SADEL with StateCharts. StateCharts are capable of describing dynamic component behavior based on the internal and external stimuli that transform a component’s current state to a new state. Similar research has been done by Richardson et al. [4,5]. However, they mainly focused on component interfaces. We have gone further by describing component operations expressed via C2SADEL invariants and pre- and post-conditions. We have also provided mechanisms to model connectors using StateCharts, thus rendering the specification more complete. As a by-product, we have also provided capabilities to ensure the consistency of the C2SADEL model with its StateCharts counterpart. Given the known (e.g., STATEMATE [8]) techniques for generating system implementations from StateCharts models, this gets us one step closer to ensuring the consistency of a system’s architecture with its implementation.

As initially proposed by Harel, StateCharts extend the conventional formalism of finite-state machines with concepts such as hierarchy and concurrency [7]. StateCharts model systems using their *states* and the *transitions* among the states. State machines are the primary means for capturing complex dynamic behavior. A state machine is an abstract machine that defines a set of conditions of existence named *states*, a set of *events* that cause pre-defined changes in states, and a set of behaviors or *actions* performed in each of these states as a result of the occurrence of events.

State machines are hierarchical in their nature; each state machine may contain several others internally. Besides hierarchy, concepts such as concurrency are extensively used in our modeling. Simple and composite states, transitions, events, actions, guards on events, and state entry variables are among other state machine

elements that we have used to build the bridge between the StateCharts and C2SADEL models.

We have developed a set of rules to be used as a guideline for creating StateCharts that describe the (dynamic) behavior of components and connectors based on their (static) C2 specification. The rules are designed based on the details of the C2 architectural style and its accompanying C2SADEL ADL. Additionally, we believe that with minimal modification the rules can be adjusted to other component and connector based architectural styles. In Table 1 a brief summary of these rules are presented. In the rest of this section we describe the complete rule system that shows how we relate the architectural elements between the C2SADEL and StateCharts models in the context of the TDS example.

5.1. Architecture

The distinct building blocks of a software architecture are components, connectors and architectural configurations (topologies) [17]. Each of them may be hierarchically constructed using other components, connectors, and configurations. State machines can be used to describe the behavior of components and connectors in the system; however, the topology of those components and connectors cannot be explicitly specified in StateCharts but must be specified in an ADL (e.g., C2SADEL) [17].

Table 1. Mapping between C2SADEL and StateCharts elements

C2SADEL	StateCharts
Architecture	Composite state machine
Components	State machine
Connectors	State machine
State variables	State’s variable
Invariants	Implicit in designing the states
Required Interface/Operation	Events
Provided Interface/Operation	Actions or events
Interface/Operation parameters	Parameters on transitions
Pre-conditions	Guards
Post-conditions	Modified state entry variable

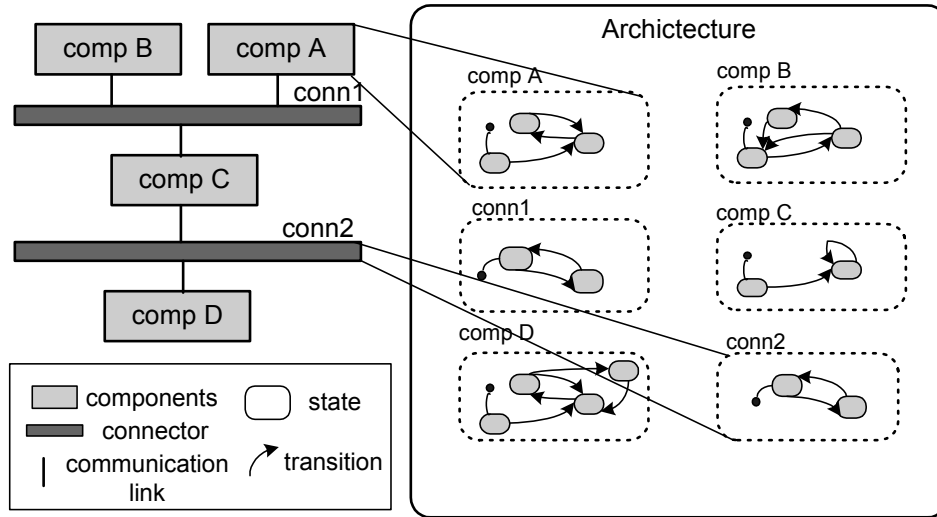


Figure 3. Example architecture in C2 style and in StateCharts

In our work, an architecture is modeled in StateCharts as a composite state machine. This state machine contains other concurrently executing state machines, each of which describes components and connectors within the architecture. The state machines of components and connectors interact through executions of events and actions. These event and action executions act as synchronizers among state machines. Figure 3 abstractly visualizes the notion of an architecture in both the C2 style and StateCharts. In the interest of clarity labels on the transitions, events, actions, parameters, and conditions have been omitted. In the context of the TDS application, the architecture of the system will be modeled using a composite state-machine called *Architecture*. Figure 4 shows side-by-side a partial model of the TDS in both C2SADEL and StateCharts.

5.2. Components

As discussed in Section 3, the C2SADEL specification of a component utilizes concepts such as state-variables, invariants, interfaces, operations, and pre- and post-conditions. We have developed a mapping between these elements and those of StateCharts.

By definition, states are conditions of existence during which some invariant condition holds [20]. These state invariants are closely related to the invariants specified in a C2SADEL description. Consequently, the states in the components' state machines should be designed, by the architect, using valid values of the state variables in the C2SADEL specification. This will ensure that invariants are reinforced in the StateCharts model. As seen in the C2SADEL specification in Figure 4 the StrategyAnalyzer component enforces conditions on the variables *status*, *row*, and *col*. The values of *row* and *col* have to be

limited to some valid pixel location, while the *status* determines whether the deployment is in an initial, requested, or done state (1, 2 and 3, respectively). Based on these invariant conditions, the states in the StrategyAnalyzer state machine must be designed by the architect such that all of these conditions hold true in those states. Changes in the values of these variables may suggest a new state. In this case, based on different values for the variable *status* we have defined three states in the corresponding state machine as shown in the top-right diagram of Figure 4.

Once the states of a component's state machine have been determined, we will need to specify the transitions among the states. These transitions correspond to stimuli received during a component's execution and cause the state of the system to change. These stimuli may originate from outside the component, resulting in changes to the component's state, or they may be initiated from within the component, which, in turn, may cause changes in the states of other components. These two situations closely resemble the way C2SADEL categorizes component services as *provided* and *required*. In C2SADEL, a required operation is instantaneous. It is an invocation to request a particular service. The provided operation is an execution of a series of steps in response to an invocation. Based on the rules of the C2 architectural style, communication between components is facilitated by connectors via asynchronous messages. These messages are either *requests* sent from a component upward or *notification* sent by a component downward. At the implementation level, a required operation roughly corresponds to an outgoing request or an incoming notification. Alternatively, provided operations roughly correspond to incoming requests or outgoing notifications. With this distinction in mind, we have

mapped C2 operations to StateCharts entities. In StateCharts *events* cause changes in the states of a state machine, while *actions* take place as a consequence of the triggering event. A StateCharts event acts as an invocation in the system and results in possibly a series of steps to be executed in response to its corresponding action.

Consequently, in our system, we model the C2SADEL's provided and required operations using *events* and *actions* in StateCharts. Required operations are modeled using events, while provided operations are mapped to either events or actions. Particularly, incoming requests become events and outgoing notifications correspond to actions. The parameters of these operations are mapped to the event and action parameters specified on transitions. An example of modeling provided and required operations is shown in Figure 4

In a C2SADEL specification, we model behavioral aspects of components' operations using pre- and post-conditions. Intuitively, pre-conditions in a C2SADEL model are mapped onto the *guards* on transitions in StateCharts. In our approach, we use a modified notion of state entry variables to express C2SADEL post-conditions in StateCharts. The standard StateCharts entry variables are defined on a per- state basis. That is, they are defined as conditions that have to hold true when the system enters the particular state due to any of its incoming transitions. Our modified entry variables are defined on a per incoming transition basis.² This enables each state to have a different entry variable specification for each transition. The details of parameters, guards and pre- and post-conditions are not shown in Figure 4 in the interest of space; however, they are summarized in Table 1.

5.3. Connectors

As with a number of architectural approaches, in the C2 style connectors are one of the essential building blocks of an architecture. In general, connectors serve as the interaction mechanism among components, and can be as simple as a procedure call or a pipe, or as complex as a Domain Name Server (DNS) or a semaphore. C2 connectors in particular propagate messages through the architecture. They may route, filter or broadcast messages. In either case, the behavior of connectors may be modeled using state machines in a manner analogous to components' behavior. In our approach connectors are described as message passing facilitators.

Figure 4 includes *Connector1* from the TDS application. Based on the C2SADEL specification, this connector propagates the *irMapInfo* message between the *StrategyAnalyzer* and the *Repository* components. This transition gets activated when the *irMapInfo* event in the triggering machine (*StrategyAnalyzer*) gets generated. This event will cause the synchronized event (*irMapInfo*) in *Connector1* to get activated, which, in turn, causes action *ipMapInfo*. The corresponding event in the *Repository* component gets triggered as a result of the connector's action. At this point the triggering of the *ipMapInfo* event in the *Repository* component results in the action *ipMapResult*, which, in turn, gets propagated through the connector and back to the *StrategyAnalyzer* component.

This series of invocations and executions corresponds to message propagation and resulting operation execution in a C2 style architecture. It is noteworthy that in this example, *Connector1* is a regular broadcast connector. To date, we have restricted our focus to message broadcast connectors as they are most common in C2-style architectures. They receive a message from a component above or below and relay it to the interested components. We intend to expand the range of connectors for modeling with StateCharts in the future. However, C2 connectors may have more sophisticated functionality such as filtering of messages and multi-cast of events. Therefore, modeling connectors as explicit entities allows us to more faithfully describe component interactions and overall system behavior.

5.4. Summary

While StateCharts define the dynamic behavior of components, C2SADEL specifies the static topology and the interconnection among components. We have developed a set of rules that relate a static component-connector based model to a dynamic state-based model. These rules can be used to aid the creation of StateCharts that adhere to the properties specified in a C2SADEL model of the system. Once we have the state-based model that is intended to extend the original component-connector based model, we have access to significant analysis capabilities that will help us in designing the system more accurately and making the transition from architecture to implementation smoother. StateCharts developed based on our approach may serve in two different ways: they may be used as a specification tool for component behavior and can help in generating more complete code from the specification; they may also be used as a testing mechanism to ensure correct run-time system behavior with regard to the specification.

² Numerous modifications of this kind have been made to Harel's original StateCharts notation (e.g., UML, STATEMATE).

```

Component StrategyAnalyzer is {
  state {
    status:Integer;
    theRow: Integer;
    theCol: Integer;
    theType:Integer;
    valid :Boolean;
    ...
  }
  invariant {
    (status > 0) \and (status < 4)
    \and (theRow < 64) \and (theCol < 64)
  }
  interface {
    req irMapInfo : MapInfo(id: String);
    req irMapResult : MapInfo(): FILE;
    prov ipUpdateData:UpdateData(row:Integer;
    col:Integer; type:Integer);
    ...
  }
  operations {
    req orMapInfo: {
      let i: STATE VARIABLE;
      pre (i <> null);
    }
    req orMapResult: {
      let PH: STATE VARIABLE;
      post (\result = PH);
    }
    prov opUpdateData: {
      let r: Integer;
      c: Integer;
      t: Integer;
      post (theRow = r)\and (theCol = c)
      \and (theType = t);
    }
    ...
  }
}
map {
  irGetMapInfo -> orGetMapInfo();
  irGetMapResult -> orGetMapResult(id -> i);
  ipUpdateData -> opUpdateData(row -> r,
  col ->c, type ->t);
  ...
}
}

```

```

Component Repository is {
  state {
    theMap: File;
    theRow: Integer;
    theCol:Integer;
    theID: String;
    ...
  }
  invariants {
    (theRow < 64) \and (theCol < 64)
  }
  interface {
    prov ipMapInfo : MapInfo (id: String);
    prov ipMapResult : MapResult (): FILE;
    ...
  }
  operations {
    prov opMapInfo:{
      let i : String;
      pre (theID <> null);
    }
    prov opMapResult:{
      post (\result = theMap);
    }
    ...
  }
  map {
    ipMapInfo -> opMapInfo(id -> i);
    ipMapResult -> opMapResult();
    ...
  }
}

```

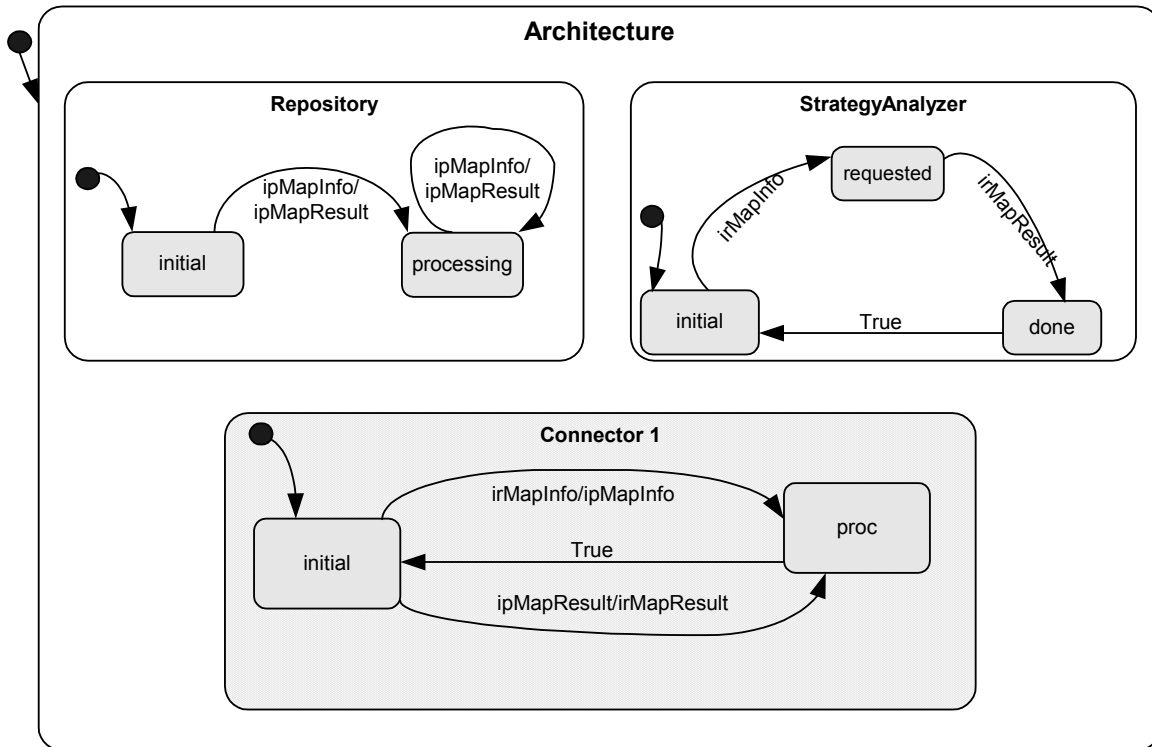


Figure 4. C2SADEL and StateCharts models of a subset of the TDS architecture

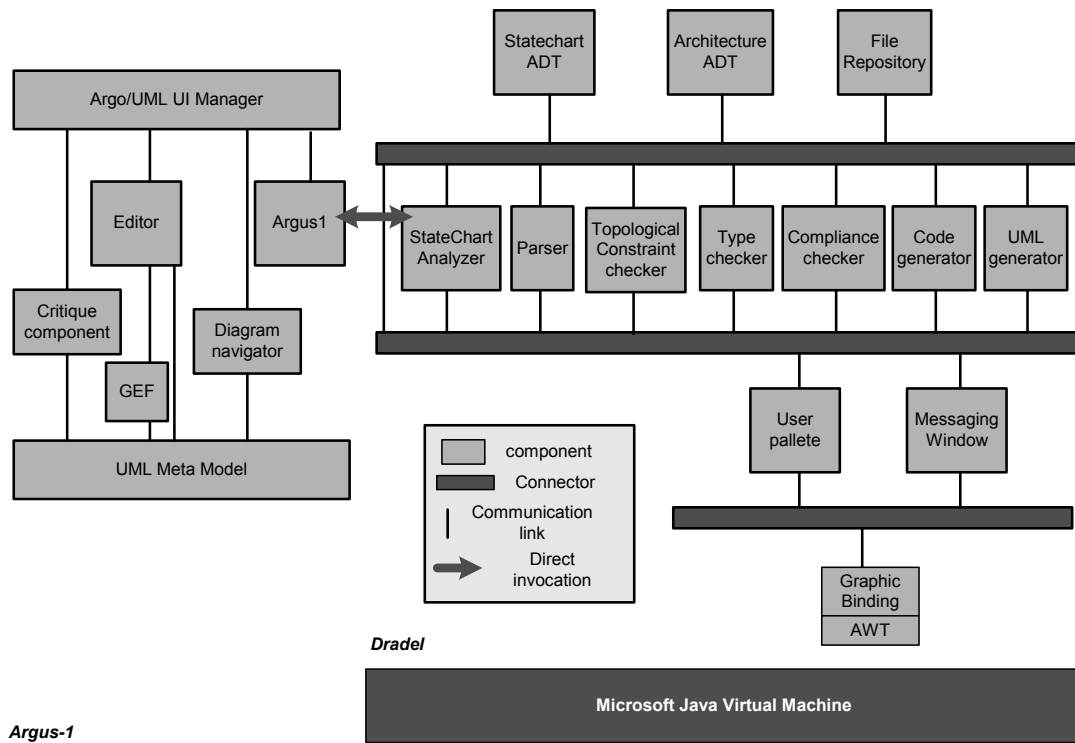


Figure 5. Integrated Environment's architecture

6. Tool Support

To automatically establish the consistency of the C2SADEL and StateCharts models, we have developed a prototype tool. This tool is an integration of Argus-I, a third-party environment for the creation and analysis of StateCharts [4,5], with DRADEL, an environment for supporting architecture-based development using C2SADEL [16]. The architecture of the tool is shown in Figure 5. Argus-I is an analysis tool built based on the framework produced by Argo/UML [23]. Argo/UML provides capabilities for drawing the variant of StateCharts supported by UML. Argus-I has extended Argo/UML by adding C2 style semantics to its StateCharts model. However, as discussed above, the part of the C2 semantics used by Argus-I is not sufficient for our modeling purposes, as Argus-I primarily leverages component interfaces, but does not focus on their behaviors nor does it consider connectors. We have extended Argus-I's model by including component invariants, operations, and pre- and post-conditions, as well as explicit connector models. This enables us to leverage Argus-I's structural and behavioral analysis, from type checking to model checking and simulation. In addition, DRADEL enables us to analyze a C2SADEL specification to ensure conformance to the C2-style syntactic and semantic rules. The integrated tool is

capable of checking the consistency of the two models in addition to static and dynamic analyses mentioned.

The process of consistency checking of the models is two fold. First, we use Argus-I to create a StateCharts model and analyze it for syntactic correctness. This includes checking for non-determinism, conflicting states, and conflicting transitions, as well as making sure all transitions are properly attached to the corresponding states. We have augmented Argus-I with a component that creates an internal representation of the StateCharts model of the system. The format of this internal representation is such that it can be directly exported into DRADEL. Separately, DRADEL parses and analyzes a C2SADEL specification of the same system. The two models are then analyzed against one another using DRADEL's consistency checking facility and any mismatches between them are identified. Figure 6 shows the user interface of the integrated tool along with an identified mismatch between the C2SADEL model of the TDS application and its StateCharts counter-part. In this particular mismatch example, the post conditions of a single operation in the two models were inconsistent.

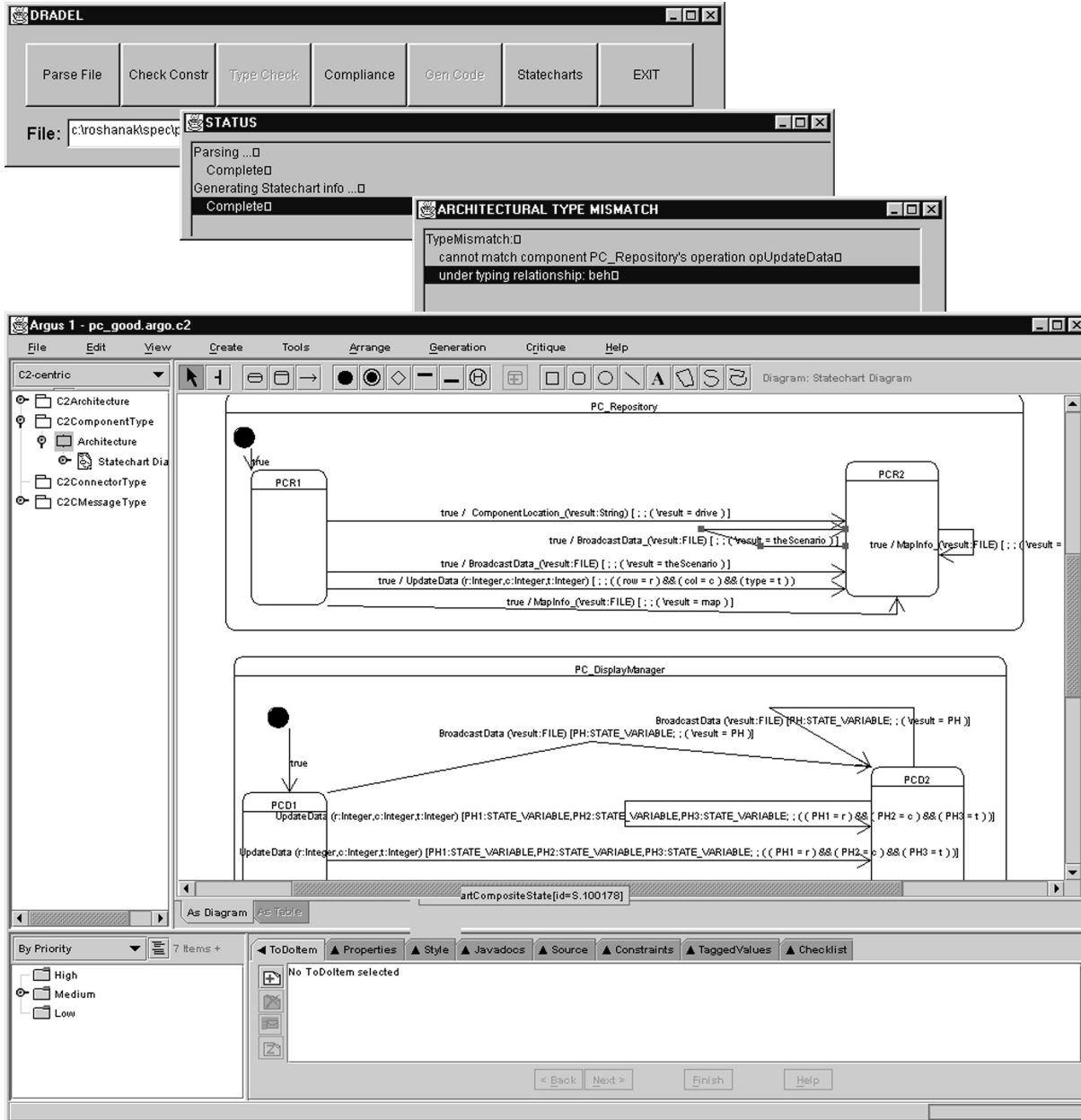


Figure 6. Integrated environment's user interface

7. Conclusion and Future Work

Architecture-based software development seeks to improve the design and development of large, complex, multi-lingual, multi-platform, and long-running systems. However, to achieve this goal specific techniques for formal modeling and specification of the systems are needed. Many ADLs have been developed mainly focusing either on static and structural aspects of the system or on the system's behavioral specification. In addition, other formal notations have been used to specify a system's behavior. For today's complex software

systems, there is a need to be able to relate the static and dynamic aspects of specifications and leverage the power of the two to perform more complete analyses of the systems under development.

In this paper we introduced one such approach for coupling static and dynamic specification. While our initial experience has been positive, we consider our results to be preliminary. We are evaluating the approach in the context of additional applications and are improving our tool support. There are also several areas that require further research. We would like to expand this work to provide guidelines for checking the

consistency of architectural models and implementation-level artifacts. In particular, we would like to study the relationship between an implementation directly generated from StateCharts, which will serve as a functionally correct prototype of the desired application, and the final system's implementation, which will leverage C2's component-based implementation infrastructure [13]. We also intend to leverage the StateCharts model and StateCharts-based implementation to aid the architecture-based testing of the application. Such testing and verification require formalizing interaction protocols among components' StateCharts and ensuring the integrity of the execution and interaction paths. Finally, we will leverage the formal type system introduced by C2SADEL [15,16] to introduce architectural types and subtyping relationships in a StateCharts model. This will enable us to expand our analysis capabilities to encompass properties of an evolving (via subtyping) system.

Acknowledgment

This material is based upon work supported by the Jet Propulsion Laboratory under Grant Number 1219801. Effort also sponsored by the National Science Foundation under Grant No. CCR-9985441. Effort also sponsored by the Defense Advanced Research Projects Agency, Rome Laboratory, Air Force Materiel Command, USAF under agreement number F30602-00-2-0615. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory or the U.S. Government. Effort also supported in part by Xerox.

References

- [1] Allen R., and Garlan D., A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3): pp. 213-249, 1997.
- [2] Batory D., and O'Malley S., The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM Transactions on Software Engineering and Methodology*, 1(4), October 1992.
- [3] Dashofy E., van der Hoek A., and Taylor R.N., A Highly-Extensible, XML-Based Architecture Description Language. To appear in *Proceedings of The Working IEEE/IFIP Conference on Software Architecture*, Amsterdam, The Netherlands, August 2001.
- [4] Dias, M., Vieira, M., Richardson, D. J., Analyzing Software architecture with Argus I. *ICSE2000 - Formal Research Demo*, June 2000, Limerick, Ireland.
- [5] Dias, M., Vieira, M., Software Architecture Analysis based on Statechart Semantics. *International Workshop on Software Specification and Design, IWSSD-10*, San Diego, CA, USA, November 2000.
- [6] Garlan D., Monroe R., and Wile D., ACME: An Architecture Description Interchange Language, in *Proceedings of CASCON'97*, November 1997.
- [7] Harel, D. Statecharts: A Visual Formalism for Complex Systems, *Science of Computer Programming*, 8, 231-274, 1987.
- [8] Harel D., Naamad A., The STATEMATE semantics of StateCharts. *ACM Transactions on Software Engineering and Methodology*, October 1996.
- [9] Hartmann J., Imoberdorf C., and Meisinger M., UML-Based Integration Testing, *presented at Int. Symp. on Software Testing and Analysis, ISSTA 00*, Portland, Oregon, USA, 2000.
- [10] Khare R., Guntersdorfer M., Oreizy P., Medvidovic N., and Taylor R.N., xADL: Enabling Architecture-Centric Tool Integration With XML. In *Proceedings of the 34th Annual Hawaii International Conference on System Sciences (HICSS-34)*, January 2001.
- [11] Luckham D. C., and Vera J., An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, vol. 21, no. 9, pp. 717-734, September 1995.
- [12] Magee J., and Kramer J., Dynamic Structure in Software Architectures, in *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 3-13, October 1996.
- [13] Medvidovic N., Oreizy P., and Taylor R.N., Reuse of Off-the-Shelf Components in C2-Style Architectures. In *Proceedings of the 1997 Symposium on Software Reusability (SSR'97)*, pp. 190-198, Boston, MA, May 17-19, 1997. Also in *Proceedings of the 1997 International Conference on Software Engineering (ICSE'97)*, pp. 692-700, Boston, MA, May 17-23, 1997.
- [14] Medvidovic N., Rosenblum D.S., Robbins J.E., and Redmiles D.F., Modeling Software Architectures in the Unified Modeling Language. To appear in *ACM Transactions on Software Engineering and Methodology*, 2001.
- [15] Medvidovic N., Rosenblum D. S., and R. N. Taylor. A Type Theory for Software Architectures. *Technical Report, UCI-ICS-98-14*, University of California, Irvine, April 1998.
- [16] Medvidovic N., Rosenblum D.S., and Taylor R.N., A Language and Environment for Architecture-Based Software Development and Evolution, in *Proceedings of the 1999 International Conference on Software Engineering*, ACM. pp. 44-53, 1999.
- [17] Medvidovic N., and Taylor R.N., A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering* 26(1), pp. 70-93, January 2000.

- [18] Mehta N., Medvidovic N., and Phadke S., Towards a Taxonomy of Software Connectors, in *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000)*, pp. 178–187, Limerick, Ireland, June 2000.
- [19] Moriconi M., Riemenschneider R.A., Introduction to SADL 1.0: a language for specifying architecture hierarchies, *Technical Report SRI-CSL-97-01*, March 1997.
- [20] Object Management Group, Unified Modeling Language specification, v1.3 spec, June 1999 (<http://www.rational.com/uml/resources/documentation/index.jsp>)
- [21] Oreizy P., Medvidovic N., and Taylor R. N., Architecture-Based Runtime Software Evolution, in *Proceedings of the 20th International Conference on Software Engineering (ICSE '98)*, pp. 177-186, Kyoto, Japan.
- [22] Perry D.E., and Wolf A.L., Foundations for the Study of Software Architectures *ACM SIGSOFT Software Engineering Notes*, vol. 17, no. 4, pp. 40-52, October 1992.
- [23] Robbins J, Argo/UML: Cognitive Support for Object-Oriented Design, version 0.6.2, April 1999 (<http://www.ics.uci.edu/pub/arch/uml/>)
- [24] Shaw, M., et al., Abstractions for Software Architecture and Tools to Support Them. *IEEE Transactions on Software Engineering*, 1995. 21(4): pp. 314-335.
- [25] Shaw M., and Garlan, D., *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.
- [26] Taylor R.N., et al., A Component- and Message-Based Architectural Style for GUI Software. *ACM Transactions on Software Engineering and Methodology*, *IEEE* 22(6), June 1996.